1. 上课须知

课程主题：ZooKeeper 第四次课 -- 源码剖析2
上课时间：20:00 - 23:00
课件休息：21:30 左右 休息10分钟
课前签到：如果能听见音乐，能看到画面，请在直播间扣 666 签到

2. 上次作业

作业：实现一个文件系统（类似于 zk 的数据模型）

验收要求：

```
1、实现基本的树形结构（DataTree），并且具备节点的增删改查等相关功能
    如果可以，帮我实现文件夹的剪切/移动
    DataTree dt = new DataTree()
    dt.insert(new DataNode())
2、具备冷启动数据恢复的功能
    具备把 内存中的数据 和 磁盘中的数据做同步和交换的 功能
    class ZKDatabase{
        private DataTree dt;
        private FileTxnSnapLog fileTxnSnapLog;
    }
    class FileTxnSnapLog{
        private TxnLog txnLog;       // 记录事务的操作日志的
        private Snapshot sh;         // 针对 DT 拍摄快照的
    }
```

3. 上次课程内容

上次课程是 ZK 的第三次课，主要讲解的是 源码解密。涉及到的主要内容：

```
第一部分：基础设施
    1、ZooKeeper版本选择
    2、Zookeeper源码环境准备
    3、ZooKeeper序列化机制
    4、ZooKeeper持久化机制
    5、ZooKeeper网络通信机制
    6、ZooKeeper的Watcher工作机制
第二部分：核心工作流程
    7、ZooKeeper的集群启动脚本分析
    8、ZooKeeper的QuorumPeerMain启动分析
    9、Zookeeper冷启动数据恢复
    10、ZooKeeper选举FastLeaderElection源码剖析
        1、准备工作
        2、选举执行
```

4. 本次ZooKeeper内容大纲

今天的主要内容是 ZK 源码解密的第二次，主要的内容有：

```
1、选举执行
2、Follower和Leader状态同步
3、创建 znode 节点源码分析
    1、会话建立
    2、节点创建
4、查询 znode 节点内容源码分析
```

5. 正式内容

5.1. ZooKeeper 选举实现源码剖析

背景知识：

```
1、所有的节点（有选举权和被选举权），一上线，就是 LOOKING 状态，当选举结束了之后，有选举权中的角色会变量另外两种：Leader, Follower。

2、需要发送选票，选票是 Vote 对象，广播到所有节点。事实上，关于选票和投票的类型有四种：
    Vote选票   Notificatioin接收到的投票   Message放入投票队列的时候会变成  ToSend待发送的选票
    还有一个特殊的中间对象：ByteBuffer  NIO 的一个API

3、当每个 zookeeper 服务器启动好了之后，第一件事就是发起投票，如果是第一次发起投票都是去找 leader，如果发现有其他 zookeeper 返回给我 leader 的信息，那么选举结束。

4、在进行选票发送的时候，每个 zookeeper 都会为其他的 zookeeper 服务节点生成一个对应的 SendWorker 和一个 ArrayBlockingQueue， ArrayBlockingQueue 存放待发送的选票，SendWorker 从队列中，获取选票执行发送。还有一个线程叫做： ReceiveWorker 真正完整从其他节点接收发送过来的投票。
```

入口：startLeaderElection(); 但是一定要记住：该方法只是为选举做准备！

大致过程如下：

```
# 入口方法
startLeaderElection();

    # 第一步：初始化选票（myid, zxid, epoch）
    currentVote = new Vote(myid, getLastLoggedZxid(), getCurrentEpoch());

    # 第二件事：初始化选举算法
    this.electionAlg = createElectionAlgorithm(electionType);

        # 第一件事：初始化了 QuorumCnxManager
        qcm = createCnxnManager();
            # 调用了 QuorumCnxManager 的构造函数 QuorumCnxManager()
            new QuorumCnxManager(.....)
                # 初始化一个 ConcurrentHashMap<Long, SendWorker> senderWorkerMap
                # 初始化一个 ArrayBlockingQueue<Message> recvQueue
                # 初始化一个 ConcurrentHashMap<Long, 发送队列>() queueSendMap
                # 初始化一个 ConcurrentHashMap<Long, ByteBuffer> lastMessageSent
                # 初始化 Listener, 这个 listener 就是  qcm 的成员变量
                listener = new Listener();

        # 第二件事：启动 QuorumCnxManager.Listener
        Listener.start()
            # 内部启动一个选举的服务端，等待其他zookeeper发送链接请求过来，建立连接，发送选
票
            # 等待有其他的zookeeper节点发送发起选举请求的链接过来
            Listener.run()
                # 启动 BIO 服务端
                ss = new ServerSocket();
                # 在初始化的时候，代码会卡在这儿
                while(!shutdown){Socket client = ss.accept();}
                    # 处理链接请求
                    receiveConnection(client);
                        # 如果对方的 myid比我小，关闭连接，只能 myid 大的向 myid 小的发
起链接
                        if(sid < this.mySid) {
                            # 重新建立连接
                            connectOne(sid);
                                # 初始化 BIO 客户端，然后初始链接
                                Socket sock = new Socket();
                                initiateConnection(sock, sid);
                                    startConnection(sock, sid);
                                        # 初始化 选票发送线程 和 投票接收线程
                                        SendWorker sw = new SendWorker(sock,
sid);
                                        RecvWorker rw = new RecvWorker(....);
                                        sw.start();
                                        rw.start();

                        }else{
                            # 处理一个链接
                            handleConnection(sock, din);
                                SendWorker sw = new SendWorker(sock, sid);
```

```
                                RecvWorker rw = new RecvWorker(sock, din, sid,
sw);
                                sw.start();
                                rw.start();
                            }

        # 第三件事：初始化选举算法
        new FastLeaderElection(this, qcm);
            starter(self, manager);
                # 第一件事： 统一发送队列
                sendqueue = new LinkedBlockingQueue<ToSend>();
                # 第二件事： 统一接收队列
                recvqueue = new LinkedBlockingQueue<Notification>();
                # 第三件事： 统一发送和接收线程！
                this.messenger = new Messenger(manager);
                    this.ws = new WorkerSender(manager);
                    ws.start()
                    this.wr = new WorkerReceiver(manager);
                    wr.start()
```

终于启动了选举：入口方法：FastLeaderElection.lookForLeader(); 这才是真正开始选举

```
# 入口，当调用 QuorumPeer 的 start() 方法的时候，它内部的 super.start() 会转到 run 方法
QuorumPeer.run()
    # 真正负责选举的是 FastLeaderElection
    FastLeaderElection.lookForLeader()
        # 初始化存储 合法选票的 容器
        HashMap<Long, Vote> recvset = new HashMap<Long, Vote>();
        # 自增逻辑时钟
        logicalclock.incrementAndGet();
        # 准备选票
        updateProposal(getInitId(), getInitLastLoggedZxid(), getPeerEpoch());
        # 广播选票
        sendNotifications();
            # 将选票存入发送队列
            sendqueue.offer(notmsg);
                # 处理选票的发送
                WorkerSender.run() + process(m);
                    manager.toSend(m.sid, requestBuffer);
                        # 将选票放置对应server的发送队列
                        addToSendQueue(bqExisting, b);
                             queue.add(buffer);  + SendWorker
                        # 发起和该服务器的链接请求，
                        connectOne(sid);
                              Socket sock = new Socket();
                              sock.connect(view.get(sid).electionAddr, cnxTO);
                              # 当我初始化一个 BIO 客户端，然后发起链接
                              # 我自己也会初始化链接(初始化SendWorker和RecvWorker )
                              # 对方也会接收到请求，然后初始化链接(初始化SendWorker和
RecvWorker)
                              initiateConnection(sock, sid);
                                    # 初始化 发送选票的 线程
                                    SendWorker sw = new SendWorker(sock, sid);
                                    # 初始化 接收投票的 线程
                                    RecvWorker rw = new RecvWorker(sock, din,
sid, sw);
                                         SendWorker.run()
```

```
                                                    # 发送选票
                                                    send(b)
                                                RecvWorker.run()
                                                    # 等接收投票
                                                    din.readFully(msgArray, 0, length);
                                                    addToRecvQueue(message)
                                                        # 将投票结果放置在接收队列recvQueue
中
                                                        recvQueue.add(msg);
```

换一种中文解释：Leader选举的基本流程如下

1. **自增选举轮次。**Zookeeper 规定所有有效的投票都必须在同一轮次中，在开始新一轮投票时，会首先对 logicalclock 进行自增操作。
2. **初始化选票。**在开始进行新一轮投票之前，每个服务器都会初始化自身的选票，并且在初始化阶段，每台服务器都会将自己推举为 Leader。
3. **发送初始化选票。**完成选票的初始化后，服务器就会发起第一次投票。Zookeeper 会将刚刚初始化好的选票放入 sendqueue 中，由发送器 WorkerSender 负责发送出去。
4. **接收外部投票。**每台服务器会不断地从 recvqueue 队列中获取外部选票。如果服务器发现无法获取到任何外部投票，那么就会立即确认自己是否和集群中其他服务器保持着有效的连接，如果没有连接，则马上建立连接，如果已经建立了连接，则再次发送自己当前的内部投票。
5. **判断选举轮次。**在发送完初始化选票之后，接着开始处理外部投票。在处理外部投票时，会根据选举轮次来进行不同的处理。

- **外部投票的选举轮次大于内部投票。**若服务器自身的选举轮次落后于该外部投票对应服务器的选举轮次，那么就会立即更新自己的选举轮次(logicalclock)，并且清空所有已经收到的投票，然后使用初始化的投票来进行PK以确定是否变更内部投票。最终再将内部投票发送出去。
- **外部投票的选举轮次小于内部投票。**若服务器接收的外选票的选举轮次落后于自身的选举轮次，那么Zookeeper就会直接忽略该外部投票，不做任何处理，并返回步骤4。
- **外部投票的选举轮次等于内部投票。**此时可以开始进行选票PK。

6. **选票PK。**在进行选票PK时，符合任意一个条件就需要变更投票。

- 若外部投票中推举的Leader服务器的选举轮次大于内部投票，那么需要变更投票。
- 若选举轮次一致，那么就对比两者的ZXID，若外部投票的ZXID大，那么需要变更投票。
- 若两者的ZXID一致，那么就对比两者的SID，若外部投票的SID大，那么就需要变更投票。

7. **变更投票。**经过PK后，若确定了外部投票优于内部投票，那么就变更投票，即使用外部投票的选票信息来覆盖内部投票，变更完成后，再次将这个变更后的内部投票发送出去。
8. **选票归档。**无论是否变更了投票，都会将刚刚收到的那份外部投票放入选票集合 recvset 中进行归档。recvset 用于记录当前服务器在本轮次的 Leader 选举中收到的所有外部投票（按照服务队的SID区别，如{(1, vote1), (2, vote2)...}）。
9. **统计投票。**完成选票归档后，就可以开始统计投票，统计投票是为了统计集群中是否已经有过半的服务器认可了当前的内部投票，如果确定已经有过半服务器认可了该投票，则终止投票。否则返回步骤4。
10. **更新服务器状态。**若已经确定可以终止投票，那么就开始更新服务器状态，服务器首选判断当前被过半服务器认可的投票所对应的Leader服务器是否是自己，若是自己，则将自己的服务器状态更新为 LEADING，若不是，则根据具体情况来确定自己是 FOLLOWING 或是 OBSERVING。

以上 10 个步骤就是 FastLeaderElection 的核心，其中步骤 4-9 会经过几轮循环，直到有 Leader 选举产生

5.2. Follower 和 Leader 状态同步

当 lookForLeader 方法结束之后，那么每个节点都可以更新自己的状态了，所有的节点，其中一个更新成为 Leader，其他的所有，都更新成为 Follower， 接下来这两种角色的 工作的 代码入口：

```
1、leader：            leader.lead()
2、follower:          follower.followerLeader()
```

里面会涉及到很多的信息的交互：

```
1、follower必须要让leader知道自己的状态：epoch zxid sid
    必须要找出谁是leader
    发起请求连接leader
    发送自己的信息给leader
    leader接收到信息，必须要返回对应的信息给 follower
2、当leader得知follower的状态了，就确定需要做何种方式的数据同步 DIFF  TRUNC  SNAP
3、执行数据同步
4、当leader接收到超过半数 follower的 ack 之后，进入正常工作状态，集群启动完成了
```

最终总结同步的方式：

```
1、DIFF 咱两一样，不需要做什么
2、TRUNC follower的zxid比leader的zxid大，所以follower要回滚
3、COMMIT  leader的zxid比follower的zxid大，发送Proposal给foloower提交执行
4、如果follower并没有任何数据，直接使用SNAP的方式来执行数据同步（直接把数据全部序列到
follower）
```

一个zk集群的启动，事实上，就是这样的一个大致的流程：

```
1、zk集群的启动，事实上是每个节点单独启动的。只不过大家同时启动而已
2、一个节点启动，首先初始化 服务端用来处理 客户端请求的 NIOServerCnxnFactory
3、启动 QuorumPeer
4、loadDataBase() 恢复磁盘数据到内存（ZKDatabase + DataTree）
5、启动 NIOServerCnxnFactory
6、执行选举 startLeaderElection() 准备选举
7、执行选举： FastLeaderElection.lookForLeader()
8、当  lookForLeader 方法执行结束，则证明能选举出leader，所以所有节点要更改自己的状态
9、更改成为 leader 状态的节点执行 leader.lead() 进入领导状态
    如果同步完成：则执行 LeaderZooKeeperServer.start()
        启动 SessionTracker
        启动 RequestProcessors（三个处理 Pre --> Sync ---> final）
10、更改成为 follower 状态的节点则执行： follower.followLeadre() 来做数据同步
    如果同步完成：则执行 LearnerZooKeeperServer.start()
        启动 SessionTracker
        启动 RequestProcessors（三个处理 Pre --> Sync ---> final）
11、这才算真正的集群启动完成
```

到此为止，讲了那些东西：

```
1、zk 的基础组件支撑：
    序列化
    持久化
    网络通信
    Watcher
2、集群启动
    冷启动数据恢复
    选举
    状态同步
3、读写流程
```

5.3. 创建 znode 节点源码分析

写数据的入口：

```
# 初始化了 ClientCnxn 的客户端通信对象
ZooKeeper zk = new Zookeeper("bigdata02:2181", 5000, watcher);

# 发送请求: Request(GetDataRequest CreateRequest DeleteRequest)
zk.create(znode路径，znode节点数据，znode节点的权限信息，znode的类型)
```

见源码注释

5.4. 查询 znode 节点内容源码分析

写数据的入口：

```
# 初始化了 ClientCnxn 的客户端通信对象，初始化 NIO 的客户端
ZooKeeper zk = new Zookeeper("bigdata02:2181", 5000, watcher);

# 发送请求： Request(GetDataRequest CreateRequest DeleteRequest)
zk.getData(znode路径，znode的state信息，watcher)
```

整个流程同 create node ， 见源码注释

6. 总结

本次 ZooKeeper 的源码分析，主要讲述的是 ZooKeeper 选举流程 和 服务端如何处理 ZooKeeper 的读写请求！